Модел. и анализ информ. систем. Т. **21**, № **6** (2014) 120–130

УДК 519.686.2

Поведенческая идентификация программ

Баклановский М. В., Ханов А. Р.

Санкт-Петербургский государственный университет,
198504, Санкт-Петербург, Петергоф, Университетский пр., дом 28

e-mail: m.baklanovsky@spbu.ru

получена 19 сентября 2014

Ключевые слова: поведенческий анализ, аномальное обнаружение, выделение шаблонов

В работе описан алгоритм выделения шаблонов переменной длины из последовательностей системных вызовов. Эти шаблоны используются для идентификации процессов – установления того, что некоторая последовательность вызовов была сгенерирована тем же самым процессом, из которого были выделены шаблоны. Существующие алгоритмы либо вычислительно сложны, либо имеют высокий уровень ложных срабатываний по сравнению со сложным и ненадежным автоматным подходом. Предложенный в работе алгоритм имеет низкую вычислительную сложность и большую точность, чем классический N-граммный алгоритм. Тесты производительности показали, что реализованный монитор системных вызовов несущественно замедляет работу операционной системы. Предложенный алгоритм после двадцатиминутного обучения способен идентифицировать за одну минуту потоки процессов с точностью 85%. Поведенческая идентификация потоков программ используется для аномального обнаружения вредоносных воздействий на систему.

1. Введение

Борьба с вредоносными программами – одна из самых острых проблем компьютерной безопасности. Соперничество технологий атаки и защиты привело к появлению Усиленных Постоянных Угроз (Tankard, 2011).

В 1987 году Дороти Деннинг предложила для борьбы с вредоносными программами анализировать потоки «записей аудита». Этот подход был развит в работах Стефани Форрест (Forrest, 1996). Для поиска вредоносных программ было предложено искать аномалии в работе программ, в пакетах сетевого трафика, в последовательности действий пользователя, потребляемых его приложениями ресурсов и профилем использования программ. Такой подход стал называться аномальным. Для борьбы с вредоносными программами чаще используется сигнатурный подход. Он заключается в анализе существующих угроз и поиске шаблонов атак: поиске

известных файлов компьютерных вирусов, опасных сетевых пакетов, блокировке известных вредоносных воздействий на систему.

С появлением новейших видов вредоносного ПО сигнатурные методы противодействия все больше показывают свою неэффективность и технологическое отставание. Они позволяют противостоять лишь известным, старым угрозам. Аномальные алгоритмы лишены этого недостатка.

В процессе своей работы для взаимодействия с ОС программы используют динамические библиотеки, которые при необходимости могут запрашивать функции ядра ОС через системные вызовы. В качестве информации о поведении программы мы будем рассматривать последовательности системных вызовов, производимые ею. Такая информация позволяет сконцентрировать внимание на наиболее важных действиях программы.

Запущенный в системе процесс проходит конечное число трасс и генерирует цепочки системных вызовов. Задача идентификации процесса состоит в том, чтобы, имея конечное множество таких цепочек для заданного процесса, построить модель, используя которую можно распознавать любые новые цепочки системных вызовов, производимые этим процессом.

Цель работы – описать алгоритм идентификации процессов и представить результаты тестирования на реальных данных.

2. Предыдущие работы

В 1996 году в работе Стефани Форрест был описан алгоритм обнаружения аномалий в сервисах Linux на основе упреждающих пар (look-ahead pairs) вызовов. (Forrest, 1996). В 1998 году был опубликован алгоритм, основанный на выделении последовательностей вызовов фиксированной длины (N-граммный метод) (Steven A. Hofmeyr, 1998). Была найдена «магическая» константа 6 – минимальная длина цепочки вызовов, при которой возможно обнаружение атаки на тестовых данных, предложенных в работе Стефани Форрест (K.Tan, 2002). Множество N-грамм может расти до огромных размеров. Современные событийно-ориентированные программы постоянно генерируют аномальные цепочки вызовов. Это плохо сказывается на уровне ложных срабатываний, а значит, и на точности алгоритма.

Был создан алгоритм выделения характерных для процесса цепочек переменной длины на основе усеченных суффиксных деревьев (H. Debar, 2000). Для последовательности строится суффиксное дерево, каждый узел помечается частотой встречаемости соответствующей подстроки. Из дерева удаляются поддеревья, частота которых меньше порога. Из оставшихся цепочек выбираются те, которые покрывают весь список системных вызовов. Но в случае перекрытия двух последовательностей часть символов будет учтена несколько раз. Алгоритм неверно оценит частоту извлекаемых им шаблонов. В результате цепочки вызовов, которые не являются достаточно частыми и характерными для последовательности, будут иметь повышенную частоту.

Для моделирования процессов использовались различные алгоритмы машинного обучения: HMM, RIPPER (Warrender Christina, 1999), нейронные сети (Anup K. Ghosh, 1999). Но все они работали несущественно лучше N-граммного алгоритма.

Важным шагом в поиске алгоритмов анализа последовательностей системных вызовов стало использование алгоритма TEIRESIAS для выделения максимальных шаблонов (Andreas Wespi, 2000). Он был адаптирован для поиска максимальных замкнутых шаблонов подпоследовательностей(D.Lo, 2008). Алгоритм TEIRESIAS работает с несколькими последовательностями. Он находит шаблоны, которые встречаются в не менее чем K последовательностях. Сначала он выделяет элементарные шаблоны, удовлетворяющие этому требованию, затем последовательно наращивает их, получая все более длинные. Алгоритм был создан для анализа цепочек ДНК. Для того, чтобы выделять шаблоны из одной цепочки системных вызовов, эта цепочка разделяется на участки фиксированной длины. Поэтому частота шаблона внутри одного такого участка не учитывается.

Автомат, вершинами которого являются списки адресов возврата, а переходы помечены номерами системных вызовов, является очень точной моделью программы (Sekar R., 2001). Язык такого автомата – это цепочки системных вызовов, которые генерирует программа. При проведении тестов авторам удавалось достичь стопроцентного обнаружения определенного класса программ. В алгоритме VtPath удалось достичь лучших результатов, чем в алгоритме с явным построением автомата (Feng, 2003).

При анализе программ алгоритмами, не использующими стек вызовов, мы пытаемся смоделировать распознаватель языка автомата, зная лишь множество его слов. Эта задача в общем случае сложна. Разворачивание стека вызовов связано с существенными накладными расходами и технически возможно не для всех программ. А вирус может подделать адреса возврата. Вместо контекста адресов возврата в алгоритмах поиска характерных цепочек для восстановления автомата используется контекст из других вызовов.

Включение дополнительной информации о вызове увеличивает точность распознавания процесса, так как позволяет уточнить его контекст. Например, существует алгоритм, который использует аргументы системных вызовов для того, чтобы разделить последовательность вызовов на подпоследовательности, которые работают с одними и теми же дескрипторами (Milea, 2012).

Наш алгоритм извлекает шаблоны переменной длины, наличие которых в потоке позволяет идентифицировать процесс, которому он принадлежит. Чтобы описать контекст вызовов, используется информация о повторениях. Предложенный в работе алгоритм обладает рядом особенностей:

1. В отличие от N-граммного алгоритма, наш алгоритм извлекает шаблоны переменной длины, что позволяет охватывать больший контекст вызовов.

2. В отличие от алгоритма с построением суффиксного дерева, наш алгоритм исключает пересечения шаблонов в последовательности, частота считается без наложений.

3. В отличие от алгоритмов построения итеративных шаблонов, наш алгоритм работает с последовательностью вызовов как с единым целым, частоты шаблонов не теряются.

3. Описание алгоритма

Дана последовательность вызовов S=(s1,..,sN). Алгоритм выделения шаблонов разбивается на 2 этапа.

3.1. Этап 1. Постоение суффиксного массива

Над алфавитом номеров системных вызовов выстраивается произвольный линейный порядок. Далее строится суффиксный массив – лексикографически отсортированный массив суффиксов S. Алгоритм построения суффиксного массива позволяет одновременно строить массив Наибольших Общих Префиксов (LCP) между каждой парой соседних суффиксов. Используя его, можно вычислить LCP между любыми двумя суффиксами массива как минимума из всех LCP суффиксов, лежащих в массиве между ними. Очевидно, каждый LCP встречается в S хотя бы два раза.

3.2. Этап 2. Извлечение шаблонов

Зафиксируем три константы:

1. MinSup – минимальное число встречаемости шаблона.
2. MinLength – минимальная длина шаблона.
3. MaxCount – максимальное количество шаблонов.

Далее мы будем использовать кортеж (MinSup, MinLength, MaxCount) для указания параметров нашего алгоритма. Создадим хеш-таблицу SIZE, ключом которой будет число L, а значением – связный список всех LCP длины L.

Далее алгоритм работает итерационно. На каждом шаге он выбирает очередной LCP наибольшей длины из таблицы SIZE, считает его частоту, отмечая его вхождения в массиве HIT.

Если этот LCP встречается не менее MinSup раз, то она включается в массив FEATURES. Псевдокод алгоритма приведен на листинге 1.

3.3. Идентификация неизвестной последовательности

Даны последовательность системных вызовов S’ и множество шаблонов FEATURES. Требуется оценить степень похожести S’ на трассу, использованную при обучении. Для этого вычислим следующие характеристики:

1. MissCount – число вызовов, не вошедших ни в один шаблон.
2. PatCount – число найденных в S’ шаблонов из FEATURES.
3. MaxPatLen – наибольшая длина найденного в S’ шаблона из FEATURES.

Чем выше PatCount и MaxPatLen и ниже MissCount, тем выше должна быть оценка степени схожести между процессами, сгенерировавшими S и S’.

Листинг 1. Псевдокод алгоритма выделения шаблонов

```
SM[0..N]–суффиксный массив
TEXT[0..N]–исходная последовательность
LCP[i] – длина наибольшего общего префикса между суффиксами в
    позициях SM[i] и SM[i+1]
HIT[0..N]–массив занятых номеров
SIZE[0..C] – массив списков
C = max(|LCP(i)|)
FEATURES=() – массив полученных шаблонов
For I in 0 to N–1:
  Push SIZE[LCP[I]],I
For I in C to MinLen:
  For E in SIZE[I]:
    (imin,imax)=(E,E);
    L=();
    While imin>1 and LCP[imin–1]>=LCP[E]:
      imin–
    While imax<N–1 and LCP[imax+1]>=LCP[E]:
      imax++
    For T in imin to imax:
      If HIT[SM[T]]==1 or HIT[SM[T]+LCP[T]–1]==1
       or HIT[SM[T+1]]==1 or HIT[SM[T+1]+LCP[T]–1]==1:
        next
      For J in SM[T] to SM[T]+LCP[T]–1:
        HIT[J]=1
      For J in SM[T+1] to SM[T+1]+LCP[T]–1:
        HIT[J]=1
      Push L, T
      If size(L) < MinFreq:
        For f in L:
          For pos in SM[f] to SM[F]+LCP[F]–1:
            HIT[pos]=0;
          For pos in SM[f+1] to SM[F+1]+LCP[F]–1:
            HIT[pos]=0;
        Next
      Symbols=TEXT[SM[E]..SM[E]+LCP[E]–1]
      Push FEATURES, Symbols
      If size(FEATURES)>=MaxCount:
        break;
```

4. Оценка сложности

Первый этап алгоритма имеет сложность O(N log N). Это сложность построения суффиксного массива в нашей реализации. Существует также алгоритм Каркkайнена–Сандерса, который строит эту структуру данных за линейное время. Массив LCP строится по алгоритму Касаи за линейное время.

Для выполнения второго этапа нужно перебрать все N-1 наибольших общих префиксов. Для каждого из них в среднем за O(log N) и в худшем случае за O(N) нужно посчитать число встречаемости. В итоге сложность второго этапа O(N log N) или в худшем случае $O(N^2)$.

Для поиска шаблонов строится недетерминированный автомат. При таком подходе алгоритм поиска шаблонов в последовательности является линейным.

5. Эксперименты

5.1. Тестовая платформа

Все тесты проводились на ОС Windows 7 (x86). Для записи логов системных вызовов был доработан драйвер, описанный в работе (Oderov, 2011). Драйвер записывает в буфер ядра фиксированной длины все системные вызовы наблюдаемых им процессов. Программа, управляющая драйвером, через фиксированные интервалы времени считывает и очищает весь буфер. В случае, если алгоритм на этапе обучения, логи системных вызовов записываются в файл для последующего анализа. В случае, если алгоритм на этапе обнаружения, полученные вызовы передаются алгоритму оценки.

5.2. Тесты производительности монитора

Был проведен тест, который позволяет оценить замедление системы, связанное с работой драйвера, который ведет запись системных вызовов. Была запущена программа rar.exe и заархивирована папка размером 50.8 мегабайт, в которой было 1275 файлов. Был активирован монитор всех вызовов всех процессов системы с записью результатов в оперативную память.

Архивация была выполнена несколько раз подряд, время выполнения: 69, 61, 25, 20 секунд. Такой разброс во временах связан с особенностью работы файлового кеша в Windows.

После этого система была перезагружена и архивация выполнена снова. Время выполнения: 54, 53, 39, 32 секунды.

Если учитывать лишь результаты первого теста, то выходит, что монитор замедляет работу программы примерно на 15%. Однако разброс во временах чтения файлов говорит о том, что операции, выполняемые ядром, зачастую имеют непредсказуемое время выполнения. И задержка, необходимая монитору на запись вызова в лог, несущественна по сравнению со временем выполнения операций ввода-вывода.

Аналогичные тесты были проведены при работе над системой NORT (Milea, 2012). Эта система замедляла работу ОС не более чем на 10%.

5.3. Тестирование алгоритма

Для сравнительной оценки точности нашего алгоритма был реализован N-граммный алгоритм (Steven A. Hofmeyr, 1998).

Важнейшей характеристикой алгоритма является его способность распознавать обычные программы, отличать одни легитимные программы от других. Для теста было запущено несколько программ: браузеры Opera, Internet Explorer, Chrome, плееры VLC, Media Player. Осуществлялся мониторинг всех системных процессов, включая explorer, winlogon, svchost, cmd. В тесте участвовало 70 запущенных процессов. В течение 20 минут при активном использовании системы было сгенерировано 10642770 системных вызовов. Эти данные использовались как обучающий набор. Далее в течение 1 минуты (672299 вызова) был записан еще один лог системных вызовов. Его потоки были использованы как тестовый набор. Наш алгоритм способен работать лишь с достаточно длинными последовательностями. Для идентификации потоков, совершающих небольшое количество вызовов, он не подходит.

Из 199 потоков исполнения длиной более 50 вызовов 6-граммный алгоритм неверно идентифицировал 41 (точность 80%). Наш алгоритм с параметрами (6, 2, 500) ошибся с 37 потоками (точность 82%). Если в случае неуспешного распознавания использовать вторую характеристику (наибольшую длину шаблона), то ошибочно идентифицируются 30 потоков (точность 85%). Короткие потоки идентифицируются хуже, чем длинные.

5.4. Обнаружение вредоносных программ

Более ранние тесты показывали, что идентифицировать процесс можно по самому длинному потоку исполнения (Khanov, 2012), при этом точность близка к 99%. На основании этого был проведен следующий эксперимент: в течение 5 секунд записывались все системные вызовы процесса, затем подбиралась модель на основе PatCount самого длинного потока, остальные потоки оценивались в этой модели. В таблице 1 приведены результаты тестирования.

Для того, чтобы выявить характерные особенности работы вредоносных программ, было запущено несколько компьютерных вирусов. Часть результатов приведена в Таблице 1. Наблюдались следующие особенности их работы:

1. Если вредоносная программа создает свой процесс, то его последовательности системных вызовов характерны либо короткие шаблоны, либо небольшое количество шаблонов.

2. Процесс вредоносной программы содержит в себе шаблоны из множества разных моделей, алгоритм идентификации путает его с несколькими легитимными процессами.

3. Вредоносные потоки, внедренные в другие процессы операционной системы, могут привести к тому, что у процесса сменится модель, которая его идентифицировала, если вредоносный поток совершает наибольшее в процессе число вызовов.

4. Скрипт-вирусы, которые используют стандартные системные утилиты, плохо выявляются, так как работа этих утилит не является аномалией в системе. Для их обнаружения требуется учитывать аргументы системных вызовов.

Таблица 1. Результаты поиска шаблонов в программах. Pname(cnt) – имя процесса и число вызовов в его длиннейшем потоке; ModCnt – число моделей, шаблоны которых были найдены в длиннейшем потоке; PCL – PatCount в длиннейшем потоке; MPLL – MaxPatLen в длиннейшем потоке; MCA – MissCount во всех потоках; PCA – PatCount во всех потоках; MPLA – MaxPatLen во всех потоках

Pname(Cnt)	ModCnt	PCL	MPLL	MCA	PCA	MPLA
chrome.exe(2612)	2	2884	170	0	85	15
chrome.exe(295)	2	235	23	0	171	29
explorer.exe(24327)	55	26128	101	265	21650	101
explorer.exe(839)	10	5096	33	10	108	7
Dropbox.exe(618)	7	618	31	1	51	15
svchost.exe(1057)	26	885	207	0	-	-
svchost.exe(2114)	3	1135	4	140	235	7
opera.exe(2623)	2	1643	69	0	1033	15
iexplore.exe(1272)	8	394	46	3	9	13
iexplore.exe(442)	8	224	11	0	199	12
csrss.exe(148)	1	1095	98	0	2942	98
lsass.exe(526)	18	244	378	0	-	-
SearchIndexer.exe(79)	6	13	49	0	2	15
DllHost.exe(85)	7	30	5	2	-	-
conhost.exe(37)	1	1	12	0	4	10
devenv.exe (1190)	25	4535	68	3	-	-
services.exe (147)	21	120	27	0	-	-
virus 1 (614)	45	82	9	27	-	-
virus 2 (458)	38	71	7	19	0	0
virus 3 (423)	40	149	16	24	-	-
virus 4 (302)	42	12	9	25	-	-
virus 5 (715)	46	38	9	104	0	0
virus 6 (772)	43	133	19	115	0	0
infected explorer.exe(215261)	6	100201	28	13	3281	13
infected iexplore.exe(1811)	46	1076	23	124	1020	42

Имея данные характеристики, можно построить дискриминантную функцию, которая различает вредоносное поведение от легитимного. Но такую функцию пришлось бы строить для каждой отдельно взятой системы со своим набором установленных программ и в зависимости от версии операционной системы. Для автоматизации этого процесса планируется применить алгоритмы искусственного интеллекта.

Вредоносные программы могут пытаться имитировать работу легитимного приложения. Для борьбы с этим нужно совершенствовать алгоритм и подбирать новые оценки. Однако тот факт, что каждая пользовательская машина защищена своей

моделью программ и своей дискриминантной функцией, ведет к тому, что у каждого пользователя выстраивается разная защита. А значит, даже в случае успешного взлома одной системы, взлом другой вовсе не гарантирован.

6. Вывод

Представленный нами алгоритм позволяет идентифицировать потоки программ с точностью до 85%. Монитор системных вызовов замедляет систему не более чем на 15%. Алгоритм позволяет отличать программы, участвовавшие в процессе обучения, от программ, которые не функционировали во время него.

Для увеличения точности идентификации потоков процесса необходимо:

1. Разработать алгоритм для идентификации коротких потоков.
2. Разработать алгоритм идентификации потоков, частично учитывающий аргументы системных вызовов.
3. Исследовать возможность дообучения существующей модели.

Для составления моделей важно, чтобы программы, участвующие в обучении, как можно полнее проявляли свой функционал. Для этого необходимы либо средства их автоматического запуска и тестирования, либо сбор информации с как можно большего числа пользовательских компьютеров, за которыми ведется активная работа.

Проведенные эксперименты показали, что алгоритм идентификации потоков позволяет выявлять вредоносную активность. В дальнейшем планируется с помощью алгоритмов машинного обучения разработать способ оценки степени угрозы потока по характеристикам, полученным с помощью нашего алгоритма. Также требуется провести более полные тесты производительности, включающие оценку замедления работы стандартных пользовательских интерактивных приложений.

Идентификация потоков исполнения необходима для реализации системы аномального обнаружения вредоносного поведения. Традиционным решением данной задачи является составление сигнатур вредоносных программ: блокировка сигнатур вредоносных активностей, поиск вредоносных участков кода в дампе процесса, исполнение файла в "песочнице" и поиск опасных воздействий в процессе его работы. Но динамичный рост числа вредоносных программ и улучшение их качества, крупнейшие компьютерные взломы последних лет, вирусы-шпионы, целевые атаки показывают, что современные промышленные средства обнаружения и защиты не справляются с нарастающей активностью со стороны злоумышленников.

Система аномального обнаружения позволит построить защиту, специфичную для конкретной пользовательской машины. Вместо поиска сотен миллионов вредоносных программ на компьютере пользователя будут идентифицироваться лишь несколько сотен его процессов. Все пользователи будут объединены в единую сеть, модели программ будут накапливаться на серверах и позволят идентифицировать процессы пользователя на основе информации, собранной всей системой. Это позволит подавлять работу как известных, так и неизвестных вредоносных программ. Разрабатываемые алгоритмы идентификации процессов будут использоваться при построении системы аномального обнаружения CODA (Baklanovsky, 2014).

Список литературы

1. *Andreas Wespi, Marc Dacier, and Herv Debar.* Intrusion Detection Using Variable-Length Audit Trail Patterns // Proceedings of the Third International Workshop on Recent Advances in Intrusion Detection. London, UK, UK: Springer-Verlag, 2000. P. 110–129.

2. *Anup K. Ghosh, Aaron Schwartzbard.* A study in using neural networks for anomaly and misuse detection // Proceedings of the 8th conference on USENIX Security Symposium. Vol. 8. Washington, D.C.: USENIX Association Berkeley, 1999. P. 141–151.

3. *D.Lo, S.Khoo* Mining patterns and rules for software specification discovery // Proceedings of the VLDB Endowment. VLDB Endowment, 2008. P. 1609–1616.

4. *Feng, Henry Hanping and Kolesnikov, Oleg M. and Fogla, Prahlad and Lee, Wenke and Gong, Weibo* Anomaly detection using call stack information // Proceedings 19th International Conference on Data Engineering. Washington, DC, USA: IEEE Computer Society, 2003. P. 62–75.

5. *Forrest, S., Hofmeyr, S.A., Somayaji, A., Longstaff, T.A.* A Sense of Self for Unix Processes // Proceeding SP '96 Proceedings of the 1996 IEEE Symposium on Security and Privacy. Washington, DC, USA: IEEE Computer Society, 1996. P. 120–128.

6. *H. Debar, M. Dacier, M. Nassehi, A. Wespi.* Fixed vs. variable-length patterns for detecting suspicious process behavior // J. Comput. Secur. IOS Press, 2000. P. 159–181.

7. *K. Tan, R. Maxion* "Why 6?" Defining the operational limits of stide, an anomaly-based intrusion detector // SP '02: Proceedings of the 2002 IEEE Symposium on Security and Privacy. Washington, DC, USA: IEEE Computer Society, 2002. P. 188–201.

8. *Milea, Narcisa Andreea and Khoo, Siau Cheng and Lo, David and Pop, Cristian* NORT: runtime anomaly-based monitoring of malicious behavior for windows // Proceedings of the Second International Conference on Runtime Verification. Berlin, Heidelberg: Springer-Verlag, 2012. P. 115–130.

9. *Sekar R., Bendre M., Dhurjati D., Bollineni P.* A Fast Automaton-Based Method for Detecting Anomalous Program Behaviors // Proceedings of the 2001 IEEE Symposium on Security and Privacy. Washington, DC, USA: IEEE Computer Society, 2001. P. 144–155.

10. *Steven A. Hofmeyr, Stephanie Forrest , Anil Somayaji* Intrusion detection using sequences of system calls // Journal of Computer Security. 1998. P. 151–180.

11. *Tankard, Colin.* Persistent threats and how to monitor and deter them. Network Security, 2011. P. 16–19.

12. *Warrender Christina, Forrest Stephanie, Pearlmutter Barak.* Detecting Intrusions Using System Calls: Alternative Data Models // IEEE Symposium on security and privacy. Oakland, CA: IEEE Computer Society, 1999. P. 133–145.

13. *Одеров Р.С., Тенсин Е.Д.* Способы размещения своего кода в ядре ОС Microsoft Windows Server 2008 // Сборник трудов межвузовской научно-практической конференции "Актуальные проблемы организации и технологии защиты информации". Санкт-Петербург: СПбНИУ ИТМО, 2011. С. 100–102. (English transl.: *Oderov R.S., Tensin Y.D.* Ways of code placing in a kernel of OS Microsoft Windows Server 2008// Proceedings of

interuniversity theoretical and practical conference "Actual problems of organization and technology of information protection". Saint-Petersburg: SPbNRU ITMO, 2011. P. 100–102).

14. *Ханов А.Р., Баклановский М.В.* Идентификация процессов программ по внешним признакам // Материалы всероссийской научной конференции по проблемам информатики "СПИСОК-2012". Санкт-Петербург: СПбГУ, 2012. С. 76–78. (English transl.: *Khanov A.R., Baklanovsky M.V.* Process identification based on external features // Proceedings of all-Russian scientific conference on Informatics problems "SPISOK-2012". Saint-Petersburg: SPbSU, 2012. P. 76–78.)

15. *Баклановский М.В., Ханов А.Р.* CODA – новая система компьютерной безопасности: обзор архитектуры системы//Материалы секции 22, XXXVIII Академические чтения по космонавтике. 2014. С. 649–650. (English transl.: *Khanov A.R., Baklanovsky M.V.* CODA – novel system for computer security: review of system architecture// Proceedings of section 22, XXXVIII Academic readings on Astronautics. Moscow, 2014. P. 649–650.)

Identification of Programs Based on the Behavior

Baklanovsky M. V., Khanov A. R.

Saint Petersburg State University,
Saint-Petersburg, Petergof, Universitetskii pr., 28, 198504, Russia

Keywords: behavior analysis, anomaly detection, pattern mining

The algorithm of pattern mining from sequences of system calls is described. Patterns are used for process identification or establishing the fact that some sequence of system calls was produced by the process which was used in pattern extraction. Existing algorithms are computationaly more complex or reveals high false positive runs in experiments in comparision with an automaton building algorithm. Our algorithm is less complex and more precise in comparision with the classical N-gram algorithm. Performance tests reveal that our kernel monitor does not significatly slow down the processing of the operating system. After 20 minutes of learning the algorithm is able to identify any thread of any process with 85% precision. Program identification based on behavior is used for anomaly detection of malicious activities in system.

Сведения об авторах:
Баклановский Максим Викторович,
Санкт-Петербургский государственный университет,
ст. преподаватель;
Ханов Артур Рафаэльевич,
Санкт-Петербургский государственный университет,
аспирант